Chapter 2

Background and State of the art

2.1 CROW

This section describes CROW, a tool tailored to create semantically equivalent variants out of a single program, either C/C++ code or LLVM bitcode. We assume that the WebAssembly programs are generated through the LLVM compilation pipeline to implement CROW. This assumption is supported by the work of Lehman et al. []; the fact that LLVM-based compilers are the most popular compilers to build WebAssembly programs [**?**] and the availability of source code (typically C/C++; and LLVM for WebAssembly) that provides a structure to perform code analysis and produce code replacements that is richer than the binary code. CROW is part of the contributions of this thesis. In Figure 2.1, we describe the workflow of CROW to create program variants.

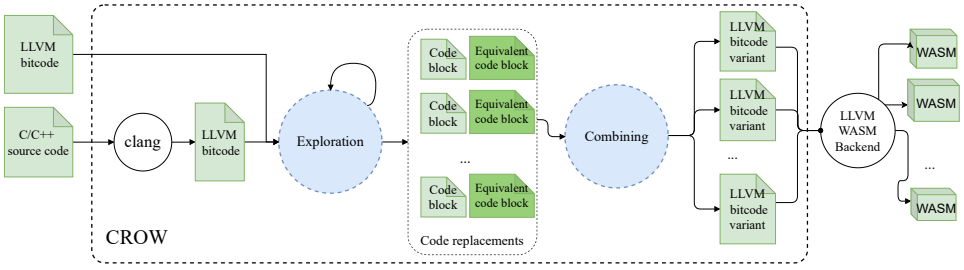

Figure 2.1: CROW workflow to generate program variants. CROW takes C/C++ source codes or LLVM bitcodes to look for code blocks that can be replaced by semantically equivalent code and generates program variants by combining them.

Figure 2.1 highlights the main two stages of the CROW's workflow, *exploration* and *combining*. The workflow starts by compiling the input program into the LLVM bitcode using clang from the source code. During the *exploration* stage, CROW takes an LLVM bitcode and, for its code blocks, produces a collection of

code replacements that are functionally equivalent to the original program. In the following, we enunciate the definitions we use along with this work for a code block, functional equivalence, and code replacement.

Definition 1 *Block (based on Aho et al. [?]): Let P be a program. A block B is a grouping of declarations and statements in P inside a function F.*

Definition 2 *Functional equivalence modulo program state (based on Le et al. [?]): Let B_1 and B_2 be two code blocks according to Definition 1. We consider the program state before the execution of the block, S_i, as the input and the program state after the execution of the block, S_o, as the output. B_1 and B_2 are functionally equivalent if given the same input S_i both codes produce the same output S_o.*

Definition 3 *Code replacement: Let P be a program and T a pair of code blocks (B_1, B_2). T is a candidate code replacement if B_1 and B_2 are both functionally equivalent as defined in Definition 2. Applying T to P means replacing B_1 by B_2. The application of T to P produces a program variant P' which consequently is functionally equivalent to P.*

We implement the *exploration* stage by retargeting a superoptimizer for LLVM, using its subset of the LLVM intermediate representation. CROW operates at the code block level, taking them from the functions defined inside the input LLVM bitcode module. In addition, the retargeted superoptimizer is in charge of finding the potential places in the original code blocks where a replacement can be applied. Finally, we use the enumerative synthesis strategy of the retargeted superoptimizer to generate code replacements. The code replacements generated through synthesis are verified, according to Definition 2, by internally using a theorem prover.

Moreover, we prevent the superoptimizer from synthesizing instructions that have no correspondence in WebAssembly for the sake of reducing the searching space for equivalent program variants. Besides, we disable all optimizations in the WebAssembly LLVM backend that could reverse the superoptimizer transformations, such as constant folding and instructions normalization.

In the *combining* stage, CROW combines the candidate code replacements to generate different LLVM bitcode variants, selecting and merging the code replacements. Then for each combination, a variant bitcode is compiled into a WebAssembly binary if requested. Finally, CROW generates the variants from all possible combinations of code replacements as the power set of all code replacements.